Spring 3-1-1979

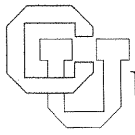
Using Data Flow Tools in Software Engineering ; CU-CS-153-79

Leon J. Osterweil
University of Colorado Boulder

Using Data Flow Tools in Software Engineering

Leon Osterweil

CU-CS-153-79



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

USING DATA FLOW TOOLS IN SOFTWARE ENGINEERING

by

Leon Osterweil
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

CU-CS-153-79 March, 1979

This work was supported by NSF Grant MCS77-02194 and Army Grant DAAG29-78-G-0046.

I. Introduction

Software engineering is a discipline which has recently been experiencing a period of considerable but unstructured growth. It now shows signs of embarking upon a phase of coordination and consolidation. There has been a large amount of work devoted to the development of software engineering tools. This seems to be particularly promising work, as tools are vehicles for capturing software engineering concepts in a way which is tangible and useful to software practitioners. Through well-implemented tools, desirable policies can be promulgated and enforced throughout a project, in a way which increases the coordination and efficiency of that project.

In the past, the quality of tools produced has been spotty. Worse, however, the goals of most tools and the domains of their efficacy have rarely been clearly enunciated. As a consequence, it has been difficult for the community of software practitioners to select tools appropriate for facilitating work on the specific tasks comprising their software development activities. Thus specification of the goals and domains of efficacy of a tool should be an important part of its documentation. The availability of such specifications should enable practitioners to intelligently select and configure a set of tools into an environment capable of supporting specific software production activities.

In this paper we propose a generic configuration of tool capabilities. We categorize many of the available tools into a few broad classes, and show how these classes have properties which are nicely complementary. We hypothesize that testing, documentation and verification are three of the most important software production activities and suggest that these activities can be nicely supported by different configurations of representatives of these few tool classes.

II. Class One - Dynamic Testing and Analysis Tools

The terms dynamic testing and dynamic analysis as used here, are intended to describe most of the systems known as execution monitors, software monitors and dynamic debugging systems ([Balz 69], [Fair 75], [Stuc 75] and [Gris 70]).

In dynamic testing systems, a comprehensive record of a single execution of a program is built. This record -- the execution history -- is usually obtained by instrumenting the source program with code whose purpose is to capture information about the progress of the execution. Most such systems implant monitoring code after each statement of the program. This code captures such information as the number of the statement just executed, the names of those variables whose values had been altered by executing the statement, the new values of these variables, and the outcome of any tests performed by the statement. The execution history is saved in a file so that after the execution terminates it can be perused by the tester. This perusal is usually facilitated by the production of summary tables and statistics such as statement execution frequency histograms, and variable evolution trees.

Despite the existence of such tables and statistics, it is often quite difficult for a human tester to detect the source or even the presence of errors in the execution. Hence, many dynamic testing systems also monitor each statement execution checking for such error conditions as division by zero and out-of-bounds array references. The monitors implanted are usually programmed to automatically issue error messages immediately upon detecting such conditions in order to avoid having the errors concealed by the bulk of a large execution history.

Some of this can be exemplified with the aid of a simple minded program. Figure 1 shows a program whose purpose is to produce the areas of rectangles and triangles having integer dimensions, when the dimensions are given as input. The program, a procedure called areas, is divided into two major functional portions. One function, implemented by procedure lookup, returns the area of the triangle or rectangle

by using a table lookup. The two dimensions input for the object are used as the first two indices into the table, a three-dimensional array, A. If the area of a rectangle is desired, the value 1 must be input with the dimensions, a value 2 indicates the area of a triangle is desired. A value 0 causes the lookup loop to terminate. The value 1 or 2 is used as the third indexing coordinate into array, A.

Array A is initialized by the second functional portion of the program implemented by the procedure init. This procedure initializes A in a somewhat indirect way, perhaps motivated by an interest in eliminating the need for multiplications.

In Figure 2 we see the same program augmented by the code necessary to monitor for two types of errors -- division by zero and out of bounds array reference. This monitor-augmented program is typical of the code which would be generated automatically by a straightforward dynamic test tool. The monitors are positioned so as to assure that any occurrence of either of the two errors will be detected immediately before it would occur in the actual execution of the program. To a human observer it is obvious that many of these probes are redundant. We shall be very much concerned with studying the forms of automated analysis necessary to suppress such probes.

Some systems ([Fair 75], [Stuc 75]) additionally allow the human program tester to create his own monitors and direct their implantation anywhere within the program.

The greatest power of these systems is derived from the possibility of using them to determine whether a program execution is proceeding as intended. The intent of the program is captured by sets of assertions about the desired and correct relation between values of program variables. These assertions may be specified to be of local or global validity. The dynamic testing system creates and places monitors as necessary to determine whether the program is behaving in accordance with asserted intent as execution proceeds.

Figure 3 shows how the example program might be annotated with assertions. These assertions are designed to capture the intent of the program and explicitly state certain non-trivial error conditions, to

which this program seems particularly vulnerable. Figure 4 shows how the code of Figure 1 might be augmented in order to test dynamically for adherence to or violation of the assertions shown in Figure 3. It should be clear from this example that dynamic assertion verification offers the possibility of very meaningful and powerful testing. With this technique, the tester can in a convenient notation specify the precise desired functional behavior of the program (presumably by drawing upon the program's design and requirements specifications). Every execution is then tirelessly monitored for adherence to these specifications. This sort of testing obviously can focus on the most meaningful aspects of the program far more shaprly than the more mechanical approaches involvinb monitoring only for violations of certain standards such as zero division or array bounds violation.

The previous paragraphs should make it clear that dynamic testing systems have strong error detection and exploration capabilities. They excel at detecting errors during the execution of a program, and also at tracing these errors to their sources. It should be observed, however, that this information is obtained only as a result of an execution occurring in response to actual program input data. The generation of this input data is the responsibility of the tester, and in many cases involves quite a significant amount of effort and insight into the program. In addition, as Figures 2 and 4 show, the instrumentation code required in order to do error monitoring is often quite large, sometimes increasing both the size and execution time of the subject program by several multiples. Perhaps more important, however, is the fact that dynamic testing systems are capable of examining only a single execution of a program, and the results obtained are not applicable to any other execution of the program. Hence, the non-occurrence of errors in a given execution does not guarantee their absence in the program itself.

From the preceding discussion it can be seen that dynamic testing is a powerful technique for detecting the presence of errors. Hence it is a powerful testing technique. Because its results are applicable only to a single execution, it cannot be used to effectively

demonstrate the absence of errors. Thus, it is not an appropriate technique for verification (i.e., the process of showing that a program necessarily behaves as intended). Furthermore, although the assertions used for dynamic verification may themselves be valuable documentation of intent, dynamic testing does not itself create useful documentation of the nature of the program itself. Finally it is important to observe that the benefits of dynamic testing can only be derived as the result of heavy expenditures of machine storage and execution time.

III. Class Two - Static Analysis Tools

In the category of static analysis tools, we include all programs and systems which infer results about the nature of a program from consideration and analysis of a complete model of some aspect of the program. An important characteristic of such tools is that they do not necessitate execution of the subject program yet infer results applicable to all possible executions.

A very straightforward example of such a tool is a syntax analyzer. With this tool the individual statements of a program are examined one at a time. At the end of this scan it is possible to infer that the program is free of syntactic errors.

A more interesting example is a tool such as FACES [Rama 75] or RXVP [Mill 74] which performs a variety of more sophisticated error scans. These tools both, for example, perform a scan to determine whether all procedure invocations are correctly matched to the corresponding definitions. The lengths of corresponding argument and parameter lists are compared, and the corresponding individual parameters and arguments are also compared for type and dimensionality agreement. By comparing every procedure invocation with its corresponding definition in this way it is possible to assure that the program is free of any possibility of such a mismatch error. Note that this analysis requires no program execution, yet produces a result applicable to all possible executions. This sort of analysis, requiring a comparison of combinations of statements, can also be used to demonstrate that a program is free of such defects as illegal type conversions, confusion of array dimensionality, superfluous labels and missing or uninvoked procedures.

Data flow analysis is a still more sophisticated form of static analysis which is based upon consideration of sequences of events occurring along the various paths through a program. As such it is capable of more powerful analytic results than combinational scans such as those just described. The DAVE system [Oste 76, Fosd 76] is a good example of such a tool. This system examines all paths originating from the start of a FORTRAN program and is capable of determining

that no path, when executed, will cause a reference to an uninitialized variable. DAVE also examines all paths originating from a variable definition and is capable of determining whether or not there is a subsequent reference to the variable. A definition not subsequently referenced is called a "dead" definition. Hence DAVE is also capable of showing that a FORTRAN program is free of dead variable definitions.

Data flow analysis is based upon examination of a flow graph model of the subject program. The flow graph of every program unit is created and its nodes are annotated with descriptions of the uses of all variables at all nodes. Nodes representing procedure invocations cannot be annotated in this way immediately. Figure 5 shows the collection of three annotated flowgraphs which would be created to represent the variable usage by the statements of the example program of Figure 1. Procedures such as init and lookup which invoke no others are completely annotated. For such procedures a data flow analyzer like DAVE would determine the presence or absence of uninitialized variable references and dead variable definitions. This can be done by using data flow analysis algorithms such as LIVE and AVAIL [Hech 76] to efficiently determine the usage patterns of the program variables along the paths leading into or out of a program node.

The precise functioning of the algorithms can be stated as follows. Suppose either of two events ref or def(*) can happen to a program variable, say x, at a program node. The algorithms determine the functions LIVE: {program nodes} → {T,F} and AVAIL: {program nodes} → {T,F} for the nodes of the graph, where these functions are defined as follows:

If n is an arbitrary program node, then LIVE(n) = T (and we say "x is live at n") if and only if there exists a path p, from the node n to another node n' such that x is ref at n' and x is not def at any node of p between n and n'. Otherwise LIVE(n) = F.

If n is an arbitrary program node, then AVAIL(n) = T (and we say

---

*The two events are usually given as gen and kill. For the sake of the clarity of this example, we prefer to use ref and def.

"x is avail at n") if and only if for each path p, from the program start node s to n there is a node n' between s and n such that x is def at n', and x is not ref at any node between n' and n. Otherwise AVAIL(n) = F.

From these definitions, it is seen that if a variable x is LIVE at the program start node then an uninitialized reference to x is possible along some path. On the other hand if x is not LIVE at the start node, then x cannot be referenced before definition for any program execution. Similarly, suppose n is a node at which x is def. Then if x is LIVE, there must be a path from n to a node at which x is ref. If x is not LIVE at n, then n represents a dead definition of x.

From this it should be apparent that a LIVE analysis for each variable in a program is capable of determining the presence or absence of these two error types in the program. ([Fosd 76] shows that AVAIL can be used to determine important variations and subcases of these errors.) This analysis of all variables can be carried out in parallel by the algorithms described in [Hech 75] using time which can ordinarily be counted on to be linear in the number of graph nodes for usual program flowgraphs.

The reader should verify that none of the variables local to procedures init and lookup, represented by the graphs in Figure 5, are live at the procedure start nodes. Moreover, there are no local variables which are both def and not live at any node. Hence there are no uninitialized variable reference or dead definition errors for these variables in these procedures. It is also important however to observe that if, for example, the variable "xk" were misspelled in statement 14, or if statement 14 were omitted, then xk would be live at node 3. This would correctly diagnose the consequent uninitialized reference at statement 17.

The analysis of the main procedure of Figure 5 can be completed after the ref and def usage of p1, p2 and p3 in statement 36 is determined. This is accomplished in the following way by studying the manner in which the parameters to procedure lookup are used by lookup. First a linear scan of the def lists for lookup's nodes is used to ascertain

which parameters are def for lookup. Next the results of a LIVE scan are examined. If a parameter pi is live at node 21, then pi is ref for procedure lookup. These annotations are then transferred position by position from the parameter list in statement 21 to the argument list in statement 36. In this way, it is determined that p1, p2, and p3 are all ref at node 36 and none of these arguments is def.

Having done this, it is possible to complete the data flow analysis of the main program, as described above.

In summary we have seen that static analysis can be used to determine the presence or absence of certain classes of errors and to produce certain kinds of program documentation. Hence it is useful as an adjunct to a testing procedure and offers weak verification capabilities. It is also useful in supplying limited forms of documentation (e.g., the input/output behavior or a procedure's parameters and global variables). There is currently ongoing research which indicates that static analysis, particularly data flow analysis, can be used to both verify and test for wider classes of errors, as well as to produce additional forms of documentation (e.g., [Tayl 79]).

Of particular interest to us here is the possibility of using static data flow analysis to suppress certain of the probes generated by dynamic assertion verification tools as part of a comprehensive test procedure. As noted earlier, many of these probes generated by dynamic test aids are redundant. Their presence adds to the size and execution time of a test run yet has no diagnostic value. Hence an automatic procedure which removes them makes testing more efficient. It also serves to focus attention on the importance of exercising the remaining probes. Sometimes it is possible to remove all the probes generated by an assertion or single error criterion. In this case, it has been de facto demonstrated that the error being tested for cannot occur, and this aspect of the program's behavior has been verified. This perspective shows how testing and verification activities can be coordinated with each other.

For a specific example of this, let us examine the program in Figure 2. We will demonstrate how the three static analysis approaches - line-by-line, combinational and data flow - can remove progressively

more error probes. It is perhaps illuminating to observe that what is being contemplated here is actually code optimization in the classical sense (e.g., see [Alle 76], [Scha 73]. We are attempting to identify and remove redundant code in some cases and to move code to more advantageous positions in other cases. Even the techniques employed are directly derivative from optimization techniques.

A straightforward line-by-line scan of the program in Figure 2 will suffice to remove several test probes. Clearly the inequality tests in statements e2, e3, e6, and e9 must always be true. Hence no more sophisticated analysis is needed to justify the removal of these probes.

A combinational examination of contiguous sequences of tests can eliminate other probes. For example, e4 and e7 contain identical tests, without any intervening flow of control or test variable alteration. Hence one of the tests can be removed. Similarly, either e10 or e13 can be removed, and either e11 or e14 can be removed. This sort of probe removal is based upon analysis that is quite similar to "peephole optimization" [Scha 73].

Additional probe removal can be justified by data flow analysis arguments. Suppose the flow graph of the program in Figure 5 were created and annotated as follows. Each node has a def list consisting of the range test occurring at that node. The ref list at a node consists of all tests referring to variables altered by a definition at this node. Thus for example we would say that $(1 \leq j \leq 20)$ is def at e5 and e11, and that $(1 \leq j \leq 20)$ is def at e1 and at e4. We would also say that $(1 \leq j \leq 20)$ and $(1 \leq j-1 \leq 20)$ are ref at 9 and 11. More details of this annotation scheme can be found in [Oste 77] and [Boll 79].

Based upon these conventions, we conclude that if a particular test predicate, p, is both def and avail at a particular node n, then the test at node n is redundant. This analysis could be used to remove the test probes at e4 and e7, as well as the probes at e19 and e22. It should be noted that this analysis is more powerful than the combinational analysis outlined above, and thus capable of justifying the removal of the probes named earlier.

Static analysis can also be used to justify the deletion of certain probes inserted in response to assertions. Note that assertion A1 in Figure 3 expands to probe statements P1,1; P1,2; P1,3; P1,4; and P1,5. Assertion A4 also expands to 5 probes in the program in Figure 4. All of these probes could be avoided if a static scan were used first to determine which (if any) of the procedure parameters were used as outputs (defs) by the procedure.

In this case static analysis can be used to remove all probes resulting from an assertion. Hence verification of the assertion can be achieved. On the other hand, we saw that many, but not all, of the subscript range checking probes can be removed by static analysis. We shall shortly show that some additional probes can be removed by using symbolic execution and constraint solving.

We have thus shown that there are significant assertion types and error categories which can be completely verified through static analysis. It seems important to determine which other assertion types and error categories give rise to probes which can be partially or totally removed by static analysis. This is currently an open research area. It is clear, however, that assertions of functional equality such as A2 and A3 are beyond easy verification by static analysis. Furthermore the removal of subscript range test probes involving functions of test variables (e.g., $1 \leq j-1 \leq 20$ in e8) seems to require either a set of special case static analyses or a different more general form of analysis. We discuss such a different type of analysis next.

IV. Class Three - Symbolic Execution Tools

By symbolic execution, we mean the process of computing the values of a program's variables as functions which represent the sequence of operations carried out as execution is traced along a specific path through the program. If the path symbolically executed is a path from a procedure start node to an output statement, then the symbolic execution will show the functions by which all of the output values are computed. The only unknowns in these functions will be the input values (either parameters in the case of an invoked procedure or read-in values when a main program is being symbolically executed).

Thus for example suppose we symbolically execute the path 1, 2, 32, 3, 4, 5, 6, 7, 8, 9, 10, 11, 10, 11 in the program shown in Figure 1. At node 8 the value of i will be given by "1", and the value of A(1,1,1) will also be given by "1". After node 10 has been executed the first time, the value of j will be given by "2", A(1,2,1) will be given by "1 + 1". The next time node 10 is symbolically executed j will be "3" and A(1,3,1) will be "1 + 1 + 1". If the path 8,9,10,11,10 is symbolically executed, then when node 8 is reached the value of i will be an unknown and hence represented by "i". The value of A(i,1,1) will likewise be represented by "i". When node 10 is reached for the first time j will receive the value "2" and A(i,2,1) will receive the value "i + i". Similarly, the next time node 10 is reached j will receive the value "3" and A(i,3,1) will receive the value "i + i + i".

A small number of symbolic execution tools has been built [Howd 78], [King 76], [Clar 76]. These tools mechanize the creation of the formulas and maintain incremental symbol tables. They employ formula simplification heuristics in an attempt to forestall the growth in size of the generated formulas and foster recognition of the underlying functional relations. (It should be noted, however, that these simplifiers do not take roundoff error into account and, therefore, may misrepresent the actual function computed by a sequence of floating-point computations). Hence a symbolic execution tool would report the value of A(i, 3, 1) after two iterations of the loop at node 9 to be "3 * i".

The foregoing discussion strongly indicates that symbolic execution is an excellent technique for documenting a program. Symbolic traces provide documentation of the actual functioning of a program along any specific path. In order to use symbolic execution as a technique for testing and verification however, it is necessary to augment the technique with a constraint solving capability.

In order to clarify this, let us begin by observing that the above described functional behavior occurs only when the given path is executed. In general, however, a given program can execute an (often infinite) variety of paths, depending upon the program's input values. The conditions under which a given path is executed can often be determined by symbolic execution and constraint solution. Consider the program given in Figure 1, as represented by the flowgraph in Figure 5. Each edge of the flowgraph can be labelled by a predicate describing the conditions under which the edge will be traversed. Thus for example the edge (7,8) is labelled "$h \geq 1$", the edge (9,10) is labelled "$b \geq 2$", (5,6) is labelled "$h \leq 20$" and edge (11,10) is labelled "$j \leq b$" (note that node 11 is assumed to represent the loop incrementation and termination test operations). Sequential control flow edges such as (8,9) and (10,11) are labelled by the predicate "true". Now clearly a given path will be executed if and only if all of the predicates attached to all of the path edges are satisfied. Unfortunately a simple textual scan will express these constraints only in terms of the variables within the statements. Thus the constraints will in general not show their underlying interrelations. If the constraints are expressed in terms of the formulas derived through symbolic execution of the path, then a set of constraints all expressed in terms of the program's input values is obtained. Any solution of this set of constraints is a set of input values sufficient to force execution of the given path.

Thus, for example, the non-trivial constraints arising from the path 3, 4, 5, 6, 7, 8, 9, 10, 11, 10, 11 are:

$h \leq 20$	from (5,6)
$b \leq 20$	from (6,7)
$h \geq 1$	from (7,8)
$b \geq 2$	from (9,10)
$3 \leq b$	from (11,10)

From this we infer that this path will be executed if and only if $3 \leq b \leq 20$ and $1 \leq h \leq 20$. Hence argument values in these ranges will force execution of the specified path.

If we were to symbolically execute the path 1, 2, 32, 3, 4, 5, 6, 7, 8, 9, 10, 11, 10, 11 then the constraints would be:

$$3 \leq 20 \leq 20$$

$$1 \leq 20 \leq 20$$

These are all satisfied, hence we can infer that the path will always be executed.

It is important to observe that some constraint systems are unsatisfiable, indicating that the path spawning them is unexecutable. We shall make important use of this shortly. No less important is the observation that the problem of determining a solution to an arbitrary system of constraints is in general unsolvable. Hence we must not expect that this potentially useful capability can be infallibly implemented.

Experimentation has indicated, however, that for an important class of programs the constraints actually generated are quite tractible [Clar 76].

Testing and verification capabilities can be achieved by attempting to solve constraints embodying error conditions and statements of intent. Thus, for example, if we create a predicate constraining the subscript i to be "$i < 1$" at statement 8, we are specifying an out-of-bounds array reference error. This constraint is clearly inconsistent with the constraint "$i \geq 1$" attached to edge (7,8). Hence it is impossible for the first array subscript at statement 8 to be below bounds. Hence we have shown that one of the tests generated in figure 2 is superfluous. A symbolic execution of a path from node 1 through node 8 will similarly show that testing i against 20 is superfluous for that path. The dynamic test for that error condition can be safely removed if it is shown that all paths through node 8 must create constraints inconsistent with "$i \geq 20$." In this example that is the case because procedure init does not alter the value of h and init is always invoked with

h = 20. These facts can be inferred from static analysis. Hence a combination of static analysis, symbolic execution and constraint solution can be used to eliminate statement e1 of Figure 2. Similar arguments can be used to eliminate statements e4, e7, e5, e8, e10, e11, e12, e13, e14, e15, e19 and e22.

Statements e8 and e15 are particularly interesting. It could be argued that static analysis is sufficient to eliminate these subscript checking probes as well. The subscripts being checked here, however, are functions of program variables. Surely static analysis rules could be devised for each of these situations, but other rules would have to be devised for other common occurrences. The result would be an inelegant mass of special procedures. A symbolic trace, on the other hand, easily shows all functional relations, and readily expresses the needed range checking tests directly in terms of the input values. Thus the symbolic execution/ constraint solving approach provides an elegant technique which avoids the need for the inelegant special-cases approach.

It is important to note that we have analytically justified the removal of virtually all subscript checking probes from the program in Figure 2. In particular, all probes inserted to check the subscripts of statements 8, 10 and 17 can be removed. Hence we have verified that these statements correctly reference array A.

Although statement e16 is a probe for a different error (division by zero) it should be apparent that the analytic technique just described can be used to show that the test embodied in e16 is also unnecessary. This error condition is expressed as the constraint "xk=o." This will be inconsistent with any constraint set arising from symbolic execution of a path through node 14. Yet static analysis will show that node 14 must always be executed prior to node e16. Hence it is verified that the division in statement 18 is always well defined.

Probes e17, e18, e20 and e21 cannot be removed, however. In fact symbolic execution of a path such as 34, 35, 36, 21, 22, 23, 24, 25 yields only the following constraints:*

*The notation ⓘ should be read as "the ⓘ^{th} value taken as input, to this path." Hence in this case ③ means "the third value read in."

③ ≠ 0 (from edge (35,36))

③ = 1 (from edge (24,25))

Thus clearly when statement 25 is encountered ③ is constrained to be 1, but ① and ② are subject to no constraints. An out-of-bounds subscript error at statement 25 could be simulated by any of the constraints i<1, i>20, j<1, or j>20. After symbolic execution these become ①<1, ①>20, ②<1 and ②>20. None of those is inconsistent with the constraints generated by consideration of path edges. Hence a solution such as

① = 0
② = 21
③ = 1

can clearly force execution of an array subscript reference error at statement 25. Thus we see that the symbolic execution/constraint solving technique is a powerful testing aid. It should be noted that the ATTEST system [Clar 76] implements most of the capabilities just described.

Perhaps the most important use of symbolic execution/constraint solution is as a technique for verifying assertions of functional relations between program variables. At the end of the previous section it was noted that verification of assertions such as A2, A3, A5 and A6 is beyond the power of the static analyzers which had been presented. We saw that static analysis is quite adept at inferring all the possible sequences of events which might arise during execution of a program, and that by comparing these with specifications of correct and incorrect sequences, testing and verification capabilities are obtained. When the statements of correct behavior are couched as predicates involving program variables, however, symbolic execution/constraint solution is most useful. This is not surprising, as symbolic execution is a technique for tracing and manipulating the functional relations between program variables.

We have already discussed the fact that the subscript references at statements 25 and 27 may cause array bounds violations. This was determined by using symbolic execution/constraint solution to demonstrate that probes P5,1 and P6,1 are not inconsistent with path induced

constraints. Thus they cannot safely be removed and assertions A5 and A6 cannot be verified.

On the other hand, these techniques can help verify the correcness of assertions A2 and A3. By using symbolic execution for the path 10,11,10, we obtain the relation

$$A(i,j,1) = A(i,j-1,1) + i$$

Viewing this as a recurrence relation whose initial condition is given by

$$A(i,1,1) = i$$

we can obtain the analytic solution

$$A(i,j,1) = j * i$$

from the theory of finite difference equations. This relation is exactly the one asserted by A2. Hence this assertion is analytically verified and need not be dynamically verified. Clearly this capability rested heavily upon being able to draw on results from finite mathematics. Cheatham has created a tool with impressive inferential capabilities of this sort [Chea 78], although the problem of determining the closed form of a recurrence is in general intractible. Also required here is the ability to recognize when two formulas are equivalent. This problem is likewise intractible in general.

Additional pitfalls of demonstrating functional equivalence are demonstrated by assertion A3. Here we easily see that symbolic execution will establish that after statement 17

$$A(i,j,2) = A(i,j,1)/2.0$$

This is mathematically equivalent to the equation

$$A(i,j,2) = 0.5*A(i,j,1),$$

and is readily recognized as being equivalent. Because of the peculiarities of floating point hardware, however, the two formulas

$$A(i,j,1)/2.0 \text{ and } 0.5*A(i,j,1)$$

will often evaluate to different values. Hence the results of symbolic verification and dynamic verification may differ.

Despite these various limitations we are encouraged to believe

that symbolic execution/constraint solution can be used to yield impressive documentation, testing and verification capabilities. Perhaps these limitations can be put in better perspective by observing that symbolic execution and constraint solution are the basic techniques used in formal verification or so called "proof of correctness" [Elsp 72]. [Lond 75], [Hant 76]. In formal verification the intent of a program must be captured totally by assertions imbedded according to the dictates of a criterion such as the Floyd Method of Inductive Assertions [Floy 67]. The correctness verification is established by symbolically executing all code sequences lying between consecutive assertions and showing that the results obtained are consistent with the bounding assertions. The consistency demonstration is generally attempted by using predicate calculus theorem provers rather than constraint solvers as discussed here. It is crucial to observe, however, that these theorem provers are subject to the same theoretical limitations discussed earlier. The undecidablility of the First Order Predicate Calculus makes it impossible to be sure whether a theorem is true or false. Hence we cannot be guaranteed of an answer to the question of whether a symbolic execution will yield results consistent with its bounding assertions. Furthermore the symbolic execution may make simplifications and transformations of real formulas which do not recreate the functioning of floating point hardware. These and similar limitations of formal verification have long been acknowledged. Yet still formal verification is rightly regarded as a useful technique capable of increasing one's confidence in the functional soundness of a program. This is exactly the sense in which the symbolic execution/constraint solution technique just discussed should be considered worthwhile.

In fact, this technique is of more worth to a practitioner than formal verification, because of its flexibility. As already observed, formal verification requires a complete, exhaustive statement of a program's intent. The technique just described focuses on attempting to justify or disprove the validity of individual assertions. This gives the practitioner the ability to probe various individual aspects of his program as he may desire. From this perspective we view formal verification as the logical, orderly culmination of a process of verifying progressively more complete assertion sets. This culmination is rarely reached due to its prohibitive costs.

V. A Strategy for Integrating Tool Capabilities

In this section we propose some ways in which the preceding classes of tools can be combined to address important software implementation objectives. It seems that in creating software the overriding goal is to create a product which demonstrably meets its current objectives and shows promise of being adaptable to meet forseeable changes in the objectives. Much research and experimentation has been devoted to studying how to achieve this goal, and much is yet to be understood. From this past work, however, certain basic needs can be clearly discussed.

Perhaps the foremost lesson learned is that software production, especially on a multi-year, multi-person scale, is a costly, complex activity requiring effective management [Brow 77], [Bloc 77]. Such effective management can only be achieved if there is sufficient visibility into the activity. This visibility enables managers and programmers alike to decide whether the project is on the way to achieving its goals, and if not what remedial action should be taken. Hence it seems that chief among the capabilities essential in guiding a software project to success are visibility into its status and ability to determine whether the behavior of the evolving product is deviating from the intended behavior. Visibility is provided by adequate documentation made centrally available by project personnel to each other and to management. Clearly it is our thesis that this process can be substantially facilitated by tools. Determining whether or not a software product is meeting its objective is clearly the goal of the testing and verification processes which, as the preceding sections suggest, can be viewed as closely coordinated activities. Here too, our thesis is that tools can be of significant help. Moreover, as the preceding sections suggest, documentation can be viewed at least in part as an activity which is preparatory to testing and verification.

A possible diagram of this view of the software production activity is shown in Figure 6. From this diagram it is clear that the activity should be greatly facilitated by automated aids to documentation, testing and verification. The preceding sections have provided a basis for seeing how such automated aids can be fashioned from a

coalition of static analysis, symbolic execution and dynamic texting aids. We now propose some details.

A complete set of program documentation must fully describe the structure and functioning of the program. Clearly such a set must describe a wide variety of aspects of the program. At present it seems that certain of these items of description must inevitably be supplied by humans. The previous sections of the paper have shown, however, that some documentation can be generated by tools. This documentation is, moreover, probably more reliably and cheaply done by such tools. In addition, if some documentation is done by tools, the remaining documentation is likely to be done more carefully by humans, thereby suggesting the possibility of greater quality and reliability.

Earlier sections of this paper suggest that static analysis tools should be used first to create such documentation as cross reference tables, variable evolution trees, and input/output descriptions of individual variables and procedures. Symbolic execution tools should be used next to create descriptions of the functional effects of executing various paths through the code. With constraint solution, a complete input/output characterization of the code can be obtained. Performance characteristics can be measured and documented with the aid of a dynamic testing tool. It is proposed that all this documentation be stored in a central data base, forming a skeleton of the complete documentation. Editors and interactive systems might be used to gather from humans such things as text descriptions of variables and procedures.

Each of the three tool classes produces a different kind of documentation. The types of documentation are only loosely related, hence the order of application of the tools can be dictated by the importance of each to the particular project. It is important to be aware, however, that static analysis is relatively inexpensive, symbolic execution is relatively expensive, constraint solution is usually quite expensive, and dynamic testing can be quite expensive if extensive elaborate test runs are done.

In a tool-assisted testing activity, the order of application of

the tools is important. We have seen that tools can be used to focus the testing effort on paths and situations which appear to be more error prone. This is done by elimination of probes which were created to test for common programming errors and for adherence to explicit assertions. We saw that many probes can be removed by application of progressively stronger (and more costly) static analysis. Some remaining probes may be removed as a result of symbolic execution/constraint solution. We saw that these probes are likely to be the more substantive ones, monitoring for adherence to asserted functional intent. Their removal constitutes significant verification, but it can be expected that the cost of this will be relatively high. Hence symbolic execution should probably be employed cautiously or not at all as a test aid.

Finally a dynamic test tool should be used to gather definite information about the existence and sources of error in the program. As already noted, testing can only show the presence of error in a test case, and even a simple program may have an infinite number of possible test cases. Hence the tool aided procedure just outlined has added importance in that it helps suggest test cases - namely those designed to exercise probes not analytically removed.

We have seen that testing and verification can be closely related activities. It is important to remember, however, that they do differ, most noticeably in their goals and placement in the software production process. Testing is the process of looking for errors. It should be viewed as an activity which occurs frequently during code production. Verification is the process of demonstrating the absence of errors. As such it should not be undertaken until and unless testing has failed to uncover errors. Thus it is a less frequent, more critical process, usually warranting greater expense and thoroughness. Our earlier discussion has shown specific ways in which verification results can be obtained as outgrowths of testing activities. We have also seen, however, that some activities provide good verification results but are likely to be relatively costly. Because verification is a less frequent, more critical activity the extra cost may well be warranted.

A verification activity should start out like the testing activity

just described. The first step is to suppress error testing probes and probes resulting from assertions. Static analysis can be used to suppress some probes, but the most significant probes probably can be removed only by symbolic execution. Verification is achieved on an assertion-by-assertion basis only when all probes generated by a single assertion have been removed. In this way stronger more complete verification can be obtained incrementally at greater cost and effort. Complete formal verification can be attempted if desired as the culmination of this process.

A final word should be said about the need for both verification and testing. It has been observed that testing cannot demonstrate the absence of errors. Hence verification should be attempted. We have also observed that the verification process has its own risks. The most important risk is that an assertion verification attempt may end inconclusively because of the failure to determine the consistency of constraints or the truth of a theorem. As already noted, this does not necessarily signify the falsity of the assertion, just that the verification attempt ended inconclusively. Another important risk is that the verification may be successful but rely implicitly upon false assumptions about the semantics of language constructs. As an example of this, we saw that symbolic executors generally make incorrect simplifying assumptions about the functioning of floating point hardward. As a result even a complete formal verification of program correctness may not completely rule out the possibility of an execution-time error. Hence it seems that both testing and verification should be considered techniques for raising the confidence of project personnel in the software product. Each is capable of bolstering confidence in its own way, and neither should be employed to the exclusion of the other.

VI. Software Lifecycle Considerations

The previous sections of this paper have established the importance of having assertions to represent the intent of a program to be documented, tested and verified. While the importance of the assertions has been established, the source of the assertions has not been discussed. In this section we propose that the assertions reasonably and naturally originate in the early requirements and design phases of the software production process. We also propose that the testing, verification and documentation techniques already described are at least partially applicable to these earlier phases.

Figure 7 is a diagrammatic view of how the software production and maintenance process might be divided into phases. It is an adaptation of the "waterfall chart" [Reif 75] which has become widely accepted as a model of those activities. The primary goal of these models is to divide software production and maintenance into definable phases and monitoring points. This division should lead to better defined criteria for judging the quality and completeness of work in progress. We shall show how this process also produces assertions and how tools can assist in the process.

The requirements definition phase of this process is the phase during which the basic needs of the software project are enunciated. These needs are to be expressed as precisely and completely as possible, but in such a manner as to not suggest or bias an algorithmic solution. One of the most effective ways to do this is to specify the required functional and performance characteristics of the proposed program. Such a specification need not and should not suggest how the functions are to be computed. These specifications should, from the perspective of this paper, be viewed as assertions of the intent which the eventual program must satisfy. Hence the eventual code assertions must be directly traceable back to these original statements of intent. We shall explore potential mechanisms for doing this shortly.

The preliminary design phase is characterized by the process of exploring possible strategies for building an algorithmic solution which satisfies the requirements specification. During this phase processing modules and data abstractions are defined, and algorithmic

processes and data flows are represented, usually hierarchical, showing, when complete, how the principal components of the algorithmic solution are decomposed into successively more detailed specifications of data and processing. In practice, such a decomposition process invariably leads to greater understanding of the problem and consequent changes in requirements. Hence the requirements and preliminary design activities should be viewed as iterative and intertwined. Together they should be considered to be the process of gaining understanding of the nature of the problem, and agreement about an acceptable approach to its solution.

From the point of view of this paper, preliminary design is important because it specifies the required functional behavior (assertions) which apply to the various components of the solution. Hence this phase begins the process of attaching successively detailed assertions to successively smaller algorithmic units. This process should terminate with the construction of code around very detailed assertions.

The detailed design phase is the phase during which the outline of the solution, established during preliminary design, is elaborated down to the level of actual specifications for code. Detailed design should not be viewed as merely an extension of the preliminary design activity. At the start of detailed design it is necessary for the designers to reorient their thinking from a problem understanding orientation to a software construction orientation. This is a crucial phase of the software production process, during which the solution elements proposed during preliminary design must be grouped and reorganized into modules and data abstractions [Parn 72] [Lisk 75]. This reorganization should be guided by the desire to clearly capture independent solution concepts in code, and to use standard interfaces to conceal the details of their implementation. The module specifications are statements of the functional behavior required in order to realize the various design concepts. Hence they are assertions. The hierarchical decompositions of the high level modular assertions analogously become assertions specifying the behavior of the submodules comprising higher level modules. The detailed design process terminates with the creation of specifications (assertions) such as those shown in Figure 3, which are so detailed that they can be met with just a few lines of code.

As already noted, one of the primary reasons for following this phased approach to software construction is that it affords obvious opportunities for observing and evaluating progress at intermediate stages. Extensive reviews are conducted at the conclusion of each phase. One of the primary goals of such review is to establish whether or not the work completed during that phase meets the objectives as enunciated at the conclusion of the previous phase. Hence the review can quite reasonably be viewed as a testing and verification procedure, using the output of the previous phase as the statement of intent.

These reviews are invariably based upon documentation and analysis done primarily by humans. It is our contention that they can be heavily supported by tools and techniques like those described earlier in this paper. In order to do this the requirements and design specifications must be stated in terms of a rigorous formalism. Some such formalisms have already been devised. Pseudo-code languages and design representation languages such as CLU [Lisk 77] are examples of rigorous formalisms for expressing detailed design. Clearly they can be parsed and subjected to certain types of semantic analysis. Virtually all forms of static analysis and symbolic execution can be carried out on them. Hence documentation can be automatically produced and some verification automatically obtained. If the detailed design and preliminary design are both complete and rigorous enough it is possible to obtain formal verification that the detailed design meets its preliminary design objectives.

It is perhaps more surprising to note that such capabilities can reasonably be expected for requirements and preliminary design specifications. Here again the prerequisite is rigor in the specification. A number of rigorous specification methodologies have been proposed (e.g., SAMM [Step 78], SADT [Ross 77], PSL/PSA [Teic 77]). All seem to be based upon a graphical representation of the requirements and/or preliminary design.

The SREM methodology [Alfo 77] is the most interesting as it is handsomely supported by the RSL/REVS family of tools [Bell 77]. RSL is a language which is used to capture a requirements/preliminary design specification and recast it into a set of objects and relations stored in a central-

ized data base. The contents of the data base can be (and is) looked upon as a collection of annotated graphs, modelling the problem and its proposed solution. The REVS system of analytic tools examines the data base and produces documentation, analysis and limited forms of verification. Each processing element in the design has as part of its specification its input/output behavior, and a functional description which may be stated as an algorithmic graph sturcture. Hence input/output behavior can be automatically documented and verified for consistency. Symbolic execution traces can be created as documentation and for the purposes of verification. It is important to note that since SREM captures both the requirements and preliminary design in a natural intertwined fashion, verification of internal consistency is tantamount to a verification that preliminary design meets requirements.

We finally are able to see where the program assertions originate. The functional descriptions attached to the various processsing elements of an RSL-like specification are the initial program assertions. If the specification technique represents the hierarchical decomposition of these elements, then at each decomposition level functional description is attached to the processing elements. As these descriptions become more algorithmic and rigorous, the possibility of rigorous and automatic verification increases. By the beginning of detailed design they have evolved into rigorous module specifications, and are certainly a suitable basis for the automatic verification approaches described earlier.

Some of the documentation, verification and testing techniques described earlier in connection with code analysis have been applied to requirements and design representations. It remains to be demonstrated that the methodology outlined in Section V and its implementation by the tools proposed can be substantially applied equally well to requirements and design. This would establish the feasibility of a single analytic methodology and tool configuration for application at all phases of the software production process.

VII. Acknowledgments

The author wishes to thank the National Science Foundation and the U. S. Army Research Office for their support of the research activities from which most of the ideas expressed here have originated. Much valuable insight was also gained while the author was on leave of absence from the University of Colorado Department of Computer Science, and employed by the Space and Military Applications Division of Boeing Computer Services Company. The ideas expressed here have been shaped by stimulating conversations with Les Wade, John Brown, Leon Stucki, Lori Clarke, Bill Howden, Bill Riddle, Dick Taylor, Larry Peters and many others. Finally, the author wishes to thank Harriet Ortiz, Mildred Farnsworth and Arlene Hunter for their obliging willingness to type the manuscript and editors Steve Muchnick and Neil Jones for their patience.

```
1       PROCEDURE AREAS;
2       DECLARE REAL A(20,20,2), INTEGER P1, P2, P3:
3          PROCEDURE INIT (H, B);
4          DECLARE INTEGER H, B, I, J, K, REAL XK;
5          IF H > 20 THEN ERROR STOP;
6          IF B > 20 THEN ERROR STOP;
7          DO FOR I = 1 TO H;
8             A(I, 1, 1) = I;
9             DO FOR J = 2 TO B;
10               A(I, J, 1) = A(I, J-1, 1) + I;
11               END;
12            END;
13         K = 2;
14         XK = 2.0;
15         DO FOR I = 1 TO H;
16            DO FOR J = 1 TO B;
17               A(I, J, K) = A(I, J, K-1) / XK;
18               END;
19            END;
20         END;
21         PROCEDURE LOOKUP (I, J, K);
22         DECLARE INTEGER I, J, K;
23         CASE;
24,25         K = 1:  PRINT "AREA OF" I, J "RECTANGLE IS" A(I, J, K);
26,27         K = 2:  PRINT "AREA OF" I, J "TRIANGLE IS" A(I, J, K);
28,29         ELSE:  PRINT "PARAMETER ERROR:  K = " K;
30            END;
31         END;
32      CALL INIT (20,20);
33      LOOP FOREVER;
34         READ P1, P2, P3;
35         IF P3 = 0 THEN STOP;
36                   ELSE CALL LOOKUP (P1, P2, P3);
37         END;
38      END;
```

FIGURE 1: An example program

```
1         PROCEDURE AREAS;
2         DECLARE REAL A(20,20,2), INTEGER P1, P2, P3;
3            PROCEDURE INIT (H,B);
4            DECLARE INTEGER H,  B, I, J, K, REAL XK;
5            IF H > 20 THEN ERROR STOP;
6            IF B > 20 THEN ERROR STOP;
7            DO FOR I = 1 TO H;
E1              IF ~(1 <= I <= 20) THEN SUBSCRIPT RANGE ERROR;
E2              IF ~(1 <= 1 <= 20) THEN SUBSCRIPT RANGE ERROR;
E3              IF ~(1 <= 1 <= 2) THEN SUBSCRIPT RANGE ERROR;
8               A(I, 1, 1) = I;
9               DO FOR J = 2 TO B;
E4                 IF ~(1 <= I <= 20) THEN SUBSCRIPT RANGE ERROR;
E5                 IF ~(1 <= J <= 20) THEN SUBSCRIPT RANGE ERROR;
E6                 IF ~(1 <= 1 <= 2) THEN SUBSCRIPT RANGE ERROR;
E7                 IF ~(1 <= I <= 20) THEN SUBSCRIPT RANGE ERROR;
E8                 IF ~(1 <= J-1 <= 20 THEN SUBSCRIPT RANGE ERROR;
E9                 IF ~(1 <= 1 <= 2) THEN SUBSCRIPT RANGE ERROR;
10                 A(I, J, 1) = A(I, J-1, 1) + I;
11                 END;
12              END;
13           K = 2;
14           XK = 2.0;
15           DO FOR I = 1 TO H;
16              DO FOR J = 1 TO B;
E10                IF ~(1 <= I <= 20) THEN SUBSCRIPT RANGE ERROR;
E11                IF ~(1 <= J <= 20) THEN SUBSCRIPT RANGE ERROR;
E12                IF ~(1 <= K <= 2) THEN SUBSCRIPT RANGE ERROR;
E13                IF ~(1 <= I <= 20) THEN SUBSCRIPT RANGE ERROR;
E14                IF ~(1 <= J <= 20) THEN SUBSCRIPT RANGE ERROR;
E15                IF ~(1 <= K-1 <= 2) THEN SUBSCRIPT RANGE ERROR;
E16                IF XK = 0 THEN  ZERODIVIDE ERROR;
17                 A(I, J, K) = A(I, J, K-1) / XK;
18                 END;
19              END;
20           END;
21           PROCEDURE LOOKUP (I, J, K);
22           DECLARE INTEGER I, J, K;
23           CASE;
24              K = 1:
E17                IF ~(1 <= I <= 20) THEN SUBSCRIPT RANGE ERROR;
E18                IF ~(1 <= J <= 20) THEN SUBSCRIPT RANGE ERROR;
E19                IF ~(1 <= K <= 2) THEN SUBSCRIPT RANGE ERROR;
25                 PRINT "AREA OF" I, J "RECTANGLE IS"  A(I, J, K);
26              K = 2:
E20                IF ~(1 <= I <= 20) THEN SUBSCRIPT RANGE ERROR;
E21                IF ~(1 <= J <= 20) THEN SUBSCRIPT RANGE ERROR;
E22                IF ~(1 <= K <= 2) THEN SUBSCRIPT RANGE ERROR;
27                 PRINT "AREA OF" I, J "TRIANGLE IS " A(I, J, K);
28,29           ELSE:  PRINT "PARAMETER ERROR:  K = " K;
30              END;
31           END;
32        CALL INIT (20,20);
33        LOOP FOREVER;
34           READ P1, P2, P3;
```

```
35          IF P3 = 0 THEN STOP;
36                    ELSE CALL LOOKUP (P1 P2, P3);
37          END;
38      END;
```

FIGURE 2: THE PROGRAM OF FIGURE 1, WITH PROBES FOR ZERO-DIVIDE AND SUBSCRIPT RANGE ERRORS INSERTED. THE PROBES SHOWN ARE THOSE WHICH WOULD BE INSERTED BY A NAIVE DYNAMIC TEST TOOL AND HAVE STATEMENT NUMBERS PRECEEDED BY THE LETTER "E".

```
1         PROCEDURE AREAS;
2         DECLARE REAL A(20,20,2), INTEGER P1, P2, P3;
3            PROCEDURE INIT (H, B);
A1           ASSERT NO SIDE-EFFECTS
4            DECLARE INTEGER H, B, I, J, K, REAL XK;
5            IF H > 20 THEN ERROR STOP;
6            IF B > 20 THEN ERROR STOP;
7            DO FOR I = 1 TO H;
8               A(I, 1, 1) = I;
9               DO FOR J = 2 TO B;
10                 A(I, J, 1) = A(I, J-1, 1) + I;
A2                 ASSERT A(I, J, 1) = I*J;
11                 END;
12              END;
13           K = 2;
14           XK = 2.0;
15           DO FOR I = 1 TO H;
16              DO FOR J = 1 TO B;
17                 A(I, J K) = A(I, J, K-1) / XK;
A3                 ASSERT A(I, J, 2) = 0.5 * A(I, J, 1);
18                 END;
19              END;
20           END;
21           PROCEDURE LOOKUP (I, J, K);
A4           ASSERT NO SIDE-EFFECTS;
22           DECLARE INTEGER I, J, K;
A5           ASSERT 1 <= I <= 20;
A6           ASSERT 1 <= J <= 20;
23           CASE;
24,25           K = 1:  PRINT "AREA OF" I, J "RECTANGLE IS" A(I, J, K);
26,27           K = 2:  PRINT "AREA OF" I, J "TRIANGLE IS" A(I, J, K);
28,29           ELSE:  PRINT "PARAMETER ERROR:  K = " K;
30              END;
31           END;
32        CALL INIT (20,20);
33        LOOP FOREVER;
34           READ P1, P2, P3;
35           IF P3 = 0 THEN STOP;
36                     ELSE CALL LOOKUP (P1, P2, P3);
37           END;
38        END;
```

FIGURE 3: THE PROGRAM OF FIGURE 1 AS IT MIGHT BE AUGMENTED BY ASSERTIONS CAPTURING THE INTENT OF THE CODE

```
1       PROCEDURE AREAS;
2       DECLARE REAL A(20,20,2), INTEGER P1, P2, P3;
3          PROCEDURE INIT (H, B);
4          DECLARE INTEGER H, B, I, J, K, REAL XK;
P1,1       DECLARE INTEGER HTEMP, BTEMP;
P1,2       HTEMP = H;
P1,3       BTEMP = B;
5          IF H > 20 THEN ERROR STOP;
6          IF B > 20 THEN ERROR STOP;
7          DO FOR I = 1 TO H;
8             A(I, 1, 1) = I;
9             DO FOR J = 2 TO B;
10               A(I, J, 1) = A(I, J-1, 1) + I
P2,1             IF A(I, J, 1) ≠ I * J THEN PRINT "ASSERTION VIOLATION AFTER
11               END;                          STATEMENT 10" A(I, J, 1), I, J;
12            END;
13         K = 2;
14         XK = 2.0;
15         DO FOR I = 1 TO H;
16            DO FOR J = 1 TO B;
17               A(I, J, K) = A(I, J, K-1) / XK;
P3,1             IF A(I, J, 2) ≠ 0.5 * A(I, J, 1) THEN PRINT  "ASSERTION VIOLATION
                                                    AFTER STATEMENT 17" A(I,
                                                    J, 2), I, J;
18               END;
19            END;
P1,4       IF H ≠ HTEMP THEN PRINT "SIDE EFFECTS VIOLATION FOR H" H, HTEMP;
P1,5       IF B ≠ BTEMP THEN PRINT "SIDE EFFECTS VIOLATION FOR B" B, BTEMP;
20         END;
21         PROCEDURE LOOKUP (I, J, K);
22         DECLARE INTEGER I, J, K;
P4,1       DECLARE INTEGER ITEMP, JTEMP, KTEMP;
P4,2       ITEMP = I;
P4,3       JTEMP = J;
P4,4       KTEMP = K;
P5,1       IF ~(1 <= I <= 20) THEN PRINT "ASSERTION VIOLATION AFTER STATEMENT 22" I;
P6,1       IF ~(1 <= J <= 20) THEN PRINT "ASSERTION VIOLATION AFTER STATEMENT 22" J;
23         CASE;
24,25         K = 1:  PRINT "AREA OF" I, J "RECTANGLE IS" A(I, J, K);
26,27         K = 2:  PRINT "AREA OF" I, J "TRIANGLE IS" A(I, J, K);
28,29         ELSE:  PRINT "PARAMETER ERROR:  K = " K;
30            END;
P4,5       IF I ≠ ITEMP THEN PRINT "SIDE EFFECTS VIOLATION FOR I" I, ITEMP;
P4,6       IF J ≠ JTEMP THEN PRINT "SIDE EFFECTS VILLATION FOR J" J, JTEMP;
P4,7       IF K ≠ KTEMP THEN PRINT "SIDE EFFECTS VIOLATION FOR K" K, KTEMP;
31         END;
32      CALL INIT (20,20);
33      LOOP FOREVER;
34         READ P1, P2, P3;
```

```
35          IF P3 = 0 THEN STOP;
36                    ELSE CALL LOOKUP (P1, P2, P3);
37          END;
38      END;
```

FIGURE 4: THE PROGRAM OF FIGURE 1 AS IT MIGHT BE AUGMENTED BY PROBES INSERTED BY AN ASSERTION CHECKING TOOL IN RESPONSE TO THE ASSERTIONS SHOWN IN FIGURE 3. THE INSERTED PROBES ARE DENOTED BY LINE NUMBERS BENNING WITH P. LINE NUMBER PI,J IS ATTACHED TO THE JTH STATEMENT GENERATED AS A RESULT OF ASSERTION AI IN FIGURE 3.

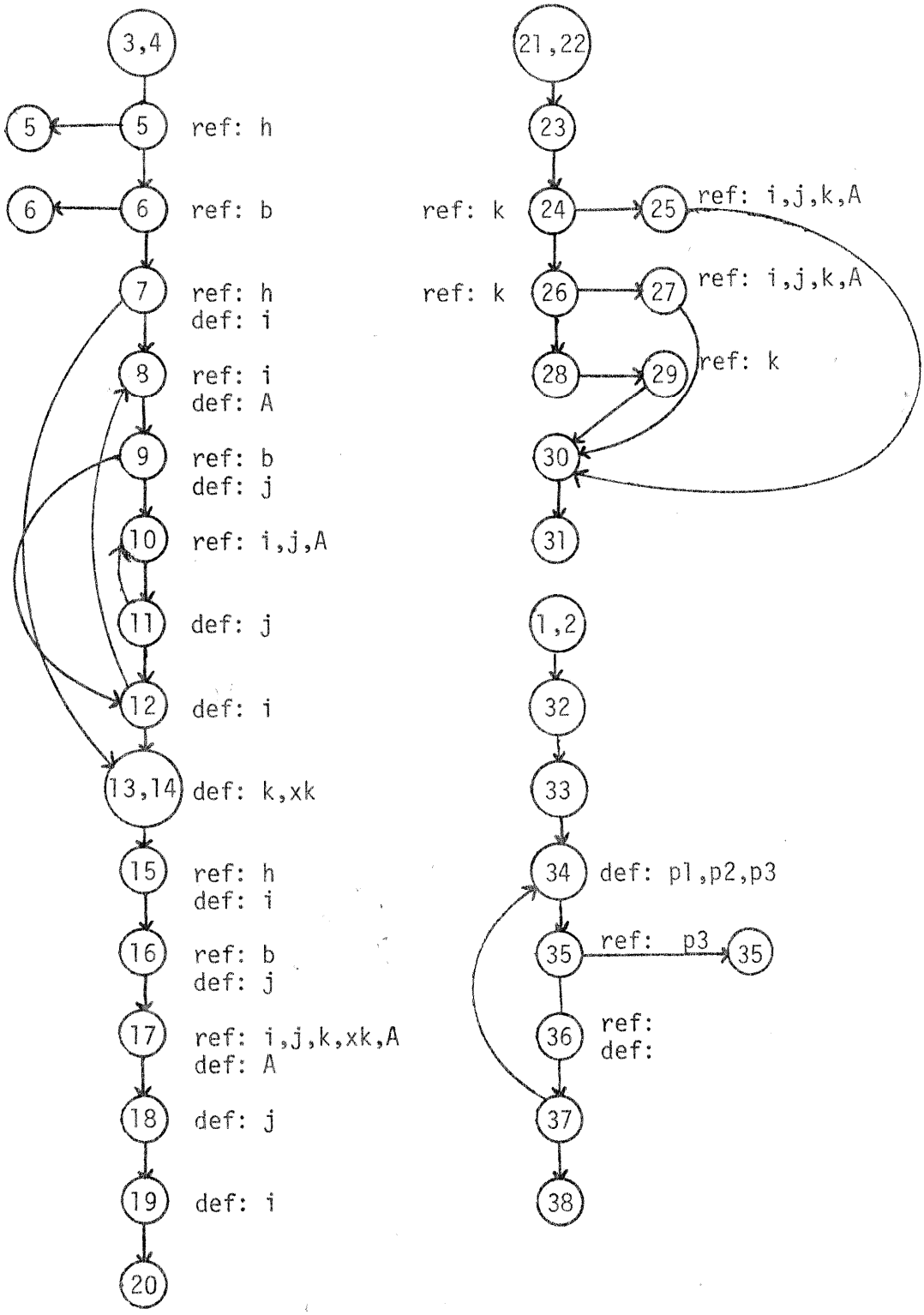


Figure 5

The flowgraphs of the three procedures in the example program of Fig. 1. The nodes are numbered by the statement of Figure 1. For each node, the program variables which are defined there and referenced there are listed. Note that node 36 represents a procedure invocation with variables as arguments. Thus the ref and def lists cannot be completed.

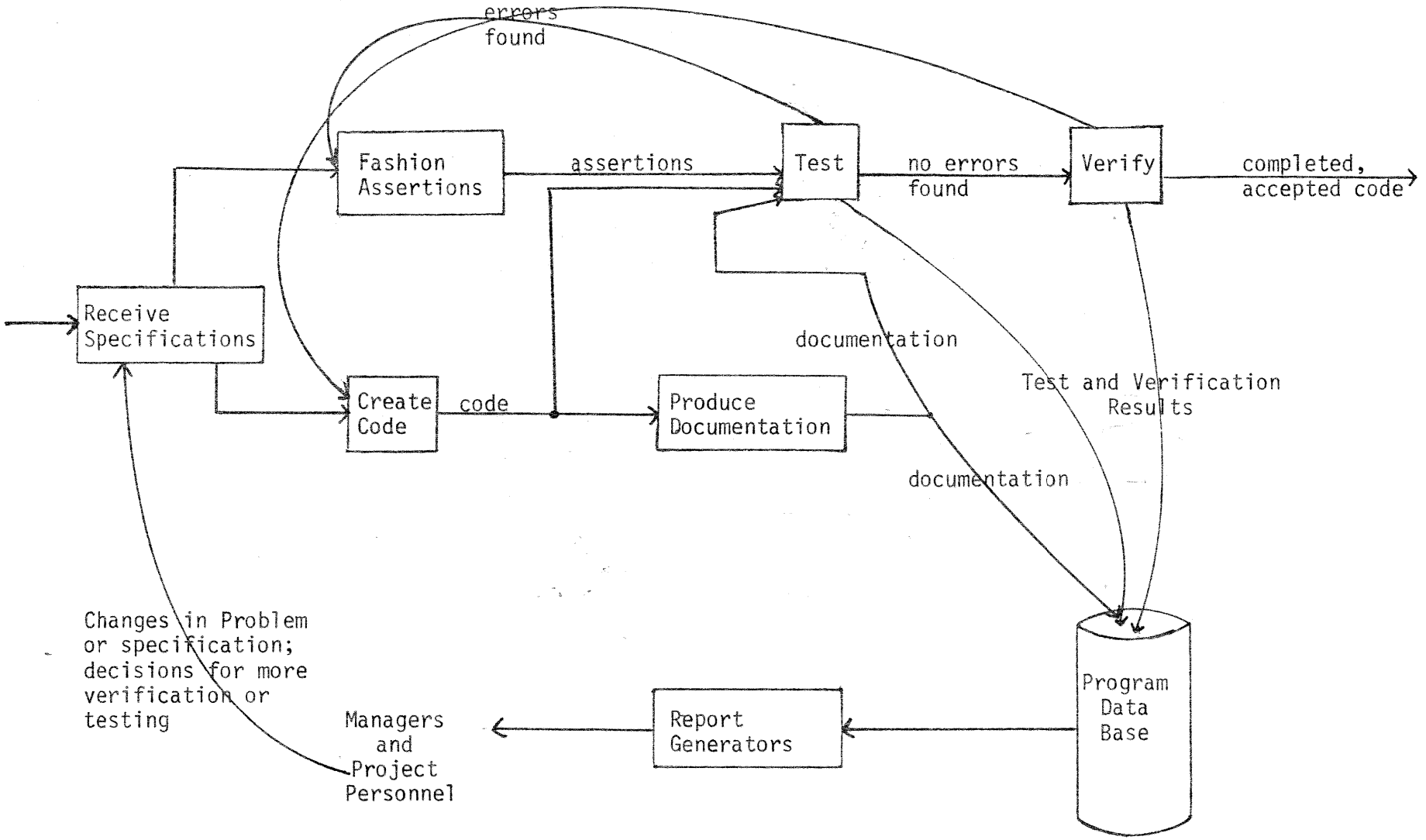


Figure 6

Suggested flow of information and processing activities in a software production project.

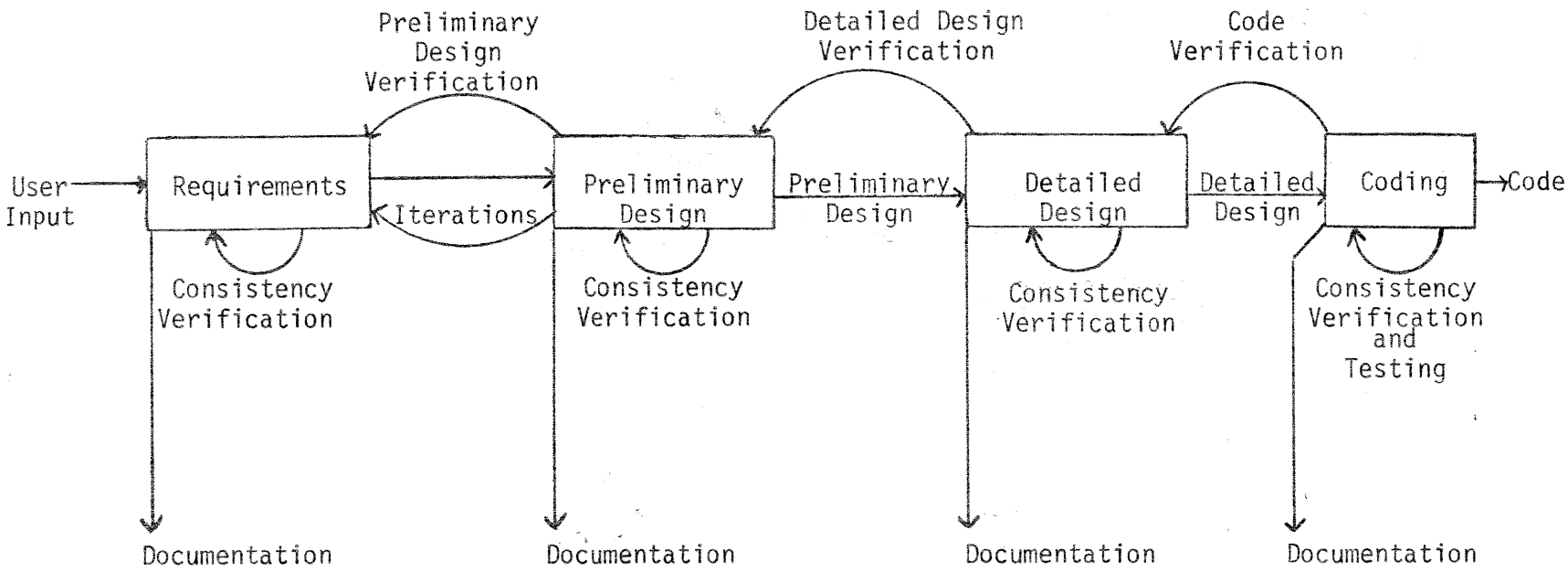


Figure 7

A view of the software production process.

References

[ALL 76] F. E. Allen and J. Cocke, "A Program DAta Flow Analysis Procedure," CACM 19, pp. 137-157 (March 1976).

[CHE 78] T. E. Cheatham, Jr. and D. Washington, "Program Loop Analysis by Solving First Order Recurrence Relations," Harvard Univ. Center for Research in Computing Technology, TR-13-78.

[CLA 76] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Trans. on Software Eng. SE-2 pp. 215-222 (Sept. 1976).

[CLE 78] G. Clemm, "The FSCAN Lexical Analyzer Generating System," Univ. of Colorado Dept. of Comp. Sci., Tech. Rpt. #CU-CS-128-78 (June 1973).

[ELS 72] B. Elspas, K. N. Levitt, R. J. Waldinger and A. Waksman, "An Assessment of Techniques for Proving Program Correctness," ACM Computing Surveys 4 pp. 97-147 (June 1972).

[FAI 75] R. E. Fairley, "An Experimental Program Testing Facility," Proc. First National Conf. on Software Engineering, IEEE Cat. #75CH0992-8C pp. 47-55 (1975).

[FLO 67] R. Floyd, "Assigning Meanings to Programs," in Mathematical Aspects of Computer Science 19 J. T. Schwartz (ed.) Amer. Math. Soc., Providence, R.I. pp. 19-32 (1967).

[FOS 76] L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability," ACM Computing Surveys 8 pp. 305-330 (Sept. 1976).

[HEC 75] M. L. Hecht and J. D. Ullman, "A Simple Algorithm for Global Data Flow Analysis Problems," SIAM J. Computing 4 pp. 519-532 (Dec. 1975).

[LON 75] R. L. London, "A View of Program Verification," 1975 International Conf. on Reliable Software, IEEE Cat. #75-CH0940-7CSR pp. 534-545 (1975).

[OST 76] L. J. Osterweil and L. D. Fosdick, "DAVE -- A Validation, Error Detection, and Documentation System for FORTRAN Programs," Software - Practice and Experience 6 pp. 473-486 (Sept. 1976).

[OST 76a] L. J. Osterweil, "A Proposal for an Integrated Testing System for Computer Programs," Univ. of Colorado Comp. Sci. Dept. Tech. Rpt. #CU-CS-093-76, (August 1976).

(Reif 75] D. J. Reifer, "Automated Aids for Reliable Software," Proc. 1975 International Conference on Reliable Software IEEE Cat. #75-CH0940-7CSR pp. 131-142 (April 1975).

[Ross 77] D. T. Ross and K. E. Schoman, Jr., "Structured Analysis for Requirement Definition," IEEE Trans. on Software Engineering SE-3 pp. 6-15 (Jan. 1977).

[Scha 73] M. Schaeffer, A Mathematical Theory of Global Program Optimization, Prentice-Hall, Englewood Cliffs, N.J. 1973.

[Step 78] S. A. Stephens and L. L. Tripp, "A Requirements Expression and Validation Tool," Proc. 3rd International Conf. on Software Eng., Atlanta, (May 1978).

[Stuc 75] L. G. Stucki and G. L. Foshee, "New Assertion Concepts in Self-Metric Software," Proceedings 1975 International Conference on Reliable Software, IEEE Cat. #75-CH0940-7CSR pp. 59-71.

[Tayl 79] R. N. Taylor and L. J. Osterweil, "Anomaly Detection in Concurrent Process Software by Static Data Flow Analysis," Univ. of Colorado at Boulder, Dept. of Comp. Sci., Tech. Rpt. #CU-CS-152-79 (April 1979).

[Teic 77] D. Teichroew and E. A. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transaction on Software Engineering SE-3 pp. 41-48 (Jan. 1977).